

Introduction to OCaml

Mattox Beckman

`beckman@iit.edu`

Illinois Institute of Technology

Objectives

§0 Objectives

Your goal in today's lecture is to gain some familiarity with OCaml. In particular, you should know...

- how to use immediate mode for interactive programming.
- the basic builtin types.
- how to use pattern matching in your functions.
- how to use tuples and lists.
- know the four ways to create variables, and how long they last.
- that **whenever you see a function, you always ask “what's its type?”**.

Language Features:

- higher order functional language
- call by value parameter passing style
- modern syntax
- parametric polymorphism
- automatic garbage collection

And two more things....

- It's very fast — the winners of the 2004, 2000 and 1999 ICFP Programming Contests used OCaml.
- The error messages make sense.

Starting OCaml Immediate Mode

§1 Introducing

- Immediate Mode statements must be terminated by a `;;`
- Notice the type inference!

```
1 $ ocaml
2      Objective Caml version 3.04
3
4 # 34 + 8;;
5 - : int = 42
6 # 27.0 +. 9.4;;
7 - : float = 36.4
8 # "hello";;
9 - : string = "hello"
```

Hello, World!

§1 Introducing

- This is the Standard First ProgramTM
- The type `unit` is like `void` in C/C++; it represents *commands*.

```
1 # print_string "Hello, world!\n";;  
2 Hello, world!  
3 - : unit = ()
```

- The type `unit` has one value, `()` (pronounced “unit” or “dot”).
- Warning: “output” has two meanings:
 - “Hello, world!” has been output (I/O) to the screen, *but*
 - the output (result) of this function is `unit`.
 - In CS 440 output will always refer to *result*.

Basic Types

§2 Types

```
1 # 20.3;;  
2 - : float = 20.3  
3 # 42;;  
4 - : int = 42  
5 # (answer = 42);;  
6 - : bool = true  
7 # "Ravi";;  
8 - : string = "Ravi"  
9 #
```

Type Errors

§2 Types

- If you try to combine things with incompatible types, you get a *type error*.
- You really need to learn how to read these.

```
1 # 1 + "hello";;  
2      ^^^^^^^  
3 This expression has type string but is here used  
4 with type int  
5 # 2.0 +. 4;;  
6      ^  
7 This expression has type int but is here used  
8 with type float  
9 #
```

- Variables behave differently in a functional language than in an imperative language!
- Variables are created, and then never changed.
- Modifications are done by *copying*.
- Very important: be able to know when a variable is created and when it is destroyed.

Variable Creation Method 1: Global Let.

```
1 # let answer=42;;  
2 val answer : int = 42  
3 # answer * 6;;  
4 - : int = 252
```

answer 42

`let name parameters = body`

```
1 # let inc x = x + 1;;
2 val inc : int -> int = <fun>
3 # inc 5;; (* Notice the arrow! *)
4 - : int = 6
5 # inc 2.4;;
6 Toplevel input:
7 # inc 2.4;;
8     ^^^
9 This expression has type float but is here used
10    with type int
```

Parameters exist *only within the function*.

Memory trace...**§3 Variables**

```
1 # let inc x = x + 1;;  
2 # inc 5;;  
3 # inc 2;;  
4 # inc 6;;
```

inc -stuff-

x 5

x 2

x 6

Each value of `x` is created when `inc` is called, and destroyed when `inc` returns.

Functions with Multiple Parameters

§3 Variables

Functions are *values* in this language!

```
1 # let triangle a b = a * a + b * b;;
2 val triangle : int -> int -> int = <fun>
3 # triangle 3 4;;
4 - : int = 25
5 # let t3 = triangle 3;;
6 val t3 : int -> int = <fun>
7 # t3 4;;
8 - : int = 25
9 # let inc = fun x -> x + 1
10 val inc : int -> int = <fun>
11 # inc 3;;
12 - : int = 4
```

Variable Creation Method 3: Local Let

§3 Variables

- You can create local variables using the `let/in` construct.

```
1 # let triangle a b =  
2     let asq = a * a in  
3     let bsq = b * b in  
4     asq + bsq;;  
5 val triangle : int -> int -> int = <fun>
```

- The variables `asq` and `bsq` are created after `triangle` is called, once the `let` expressions are reached, and destroyed once the `let` expressions are exited.

```
1 # (let x = 10 in x) + (let x = 20 in x);;  
2 - : int = 30
```

Memory Trace

§3 Variables

```
1 # let triangle a b =  
2     let asq = a * a in  
3     let bsq = b * b in  
4     asq + bsq;;  
5 val triangle : int -> int -> int = <fun>
```

triangle -stuff-

a 3

b 4

asq 9

bsq 16

Consider this code: (This is question #15 in the database.)

```
1 let x = 27;;  
2 let foo x =  
3     let x = 5 in  
4     (fun x -> print_int x) 10;;  
5 foo 12;;
```

What value will be printed?

- ☒ a) 5
- ☒ b) 10
- ☒ c) 12
- ☒ d) 27

Tuples

§3 Variables

- Tuples are one kind of *compound type*.

```
1 # let t = 2,3;;
2 val t : int * int = 2, 3
3 # let t2 = (3,42);;
4 val t2 : int * int = 3, 42
5 # let ultimate = "answer",42;;
6 val ultimate : string * int = "answer", 42
7 # let big = (2,3,"hi",4.123,"there",triangle);;
8 val big : int * int * string * float *
9           string * (int -> int -> int) =
10     2, 3, "hi", 4.123, "there", <fun>
11 #
```

```
1 # let iszero n =  
2     match n with  
3         0 -> "it's zero"  
4       | 1 -> "it's one"  
5       | _ -> "not zero or one";;  
6 val iszero : int -> string = <fun>  
7 # iszero 1;;  
8 - : string = "it's one"
```

- The *patterns* all need to have the same type.
- Also, the results....

Variable Creation Method 4: Matching

§3 Variables

- Use `match/with` to deconstruct compound types,

```
1 # let inctup a =  
2     match a with  
3         (0,y) -> y, 1  
4         | (x,y) -> x+1, y+1;;  
5 val inctup : int * int -> int * int = <fun>  
6 # inctup (2,3);;  
7 - : int * int = 3, 4  
8 # inctup (0,3);;  
9 - : int * int = 3, 1
```

A variable created with `match` lasts only in the expression after the `->`.

- A *list* can take two forms:
 - It can be an empty list, or
 - it can be an element, together with another list.
- Empty lists are written `[]`
- Non-empty lists are written `x :: xs`
 - `x` is the *head* of the list.
 - `xs` is the *tail* of the list.
- You can also write them as `[x1; x2; ...; xn]`
- Unlike tuples, lists are *monomorphic*—all the elements must have the same type.

List Examples

§4 Lists

```
1 # [];;
2 - : 'a list = []
3 # let empty = [];;
4 val empty : 'a list = []
5 # let single = [1];;
6 val single : int list = [1]
7 # let rlist = [2.3; 4.2; 5.3];;
8 val rlist : float list = [2.3; 4.2; 5.3]
9 # let badlist = [3; 4; 3.14159];;
10 Characters 21-28:
11 This expression has type float but is here used
12     with type int
```

Adding to lists

§4 Lists

● `::` adds an element to a list; `@` appends two lists together.

```
1 # let l1 = [3;6;9];;
2 val l1 : int list = [3; 6; 9]
3 # let l2 = [4;7;10];;
4 val l2 : int list = [4; 7; 10]
5 # 5 :: l1;;
6 - : int list = [5; 3; 6; 9]
7 # 10 :: 20 :: l2;;
8 - : int list = [10; 20; 4; 7; 10]
9 # l1 @ l2;;
10 - : int list = [3; 6; 9; 4; 7; 10]
```

Pattern Matching with Lists

§4 Lists

- The `match/with` construction works with lists, too.

```
1 # let getfirst l =  
2     match l with  
3         [] -> 0  
4         | x::xs -> x;;  
5 val getfirst : int list -> int = <fun>  
6 # getfirst [];;  
7 - : int = 0  
8 # getfirst [3;4;5];;  
9 - : int = 3
```

Polymorphic List Functions

§4 Lists

```
1 # let isempty l =  
2     match l with  
3         [] -> "yes"  
4         | _  -> "no";;  
5 val isempty : 'a list -> string = <fun>  
6 # isempty ["hi";"there"];;  
7 - : string = "no"  
8 # isempty [2;3];;  
9 - : string = "no"
```

- Types can be combined arbitrarily
- “Orthogonality principle” — different features don’t interfere with each other.

```
1 # let e1 = [ [20;30]; [10;5;2]; [3;6]; [] ];;  
2 val e : int list list = [[20; 30]; [10; 5; 2]; ...  
3 # let e2 = [ 2,3; 4,5; 6,7 ];;  
4 val e2 : (int * int) list = [2, 3; 4, 5; 6, 7]
```

Other syntax: if**§4 Lists**

- The `if` construct is both a *command* and an *expression*.
- The `then` and `else` branches must have the same type.

```
1 # if (3 < 5) then print_string "hi!\n" ;;
2 hi!
3 - : unit = ()
4 # let x = 10 ;;
5 val x : int = 10
6 # let y = (if x < 10 then 40 else 50) * 2 ;;
7 val y : int = 100
8 #
```


Problems 1–2**§5 Activity**

1. What will be the output of the following code?

```
1 let x = 20;;  
2 let f = fun x -> x + 1;;  
3 let y = f 30;;  
4 print_int x;;
```

2. One of the lists below is invalid. Which one?

- ☒ a) [2; 3; 4; 6]
- ☒ b) [2,3; 4,5; 6,7]
- ☒ c) [2.3,4; 3.2,5; 6,7.2]
- ☒ d) ["hi"; "there"]; ["how"]; []; ["goezit"]]

Answer 1**§5 Activity**

What will be the output of the following code?

```
1 let x = 20;;  
2 let f = fun x -> x + 1;;  
3 let y = f 30;;  
4 print_int x;;
```

Answer: 20

Answer 2**§5 Activity**

One of the lists below is invalid. Which one?

- a) `[2; 3; 4; 6]`
- b) `[2,3; 4,5; 6,7]`
- c) **Answer** `[2.3,4; 3.2,5; 6,7.2]`
- d) `[["hi"; "there"]; ["how"]; []; ["goezit"]]`

The first two elements are of type `float * int`, but the last element is of type `int * float`.

What is the type of the following functions?

```
1 let f x = x + 1  
2 let g x = x :: [1]
```

Write an OCaml function that inspects a list. If the list is empty, output "empty", otherwise, output "not empty".

What is the type of the following functions?

```
1 let f x = x + 1
2 let g x = x :: [1]
```

```
1 f : int -> int
2 g : int -> int list
```

Answer 5**§5 Activity**

Write an OCaml function that inspects a list. If the list is empty, output "empty", otherwise, output "not empty".

```
1 let isEmpty x =  
2   match x with  
3   | [] -> "empty"  
4   | _  -> "not empty"
```

Next time: The second most powerful idea in Computer Science.